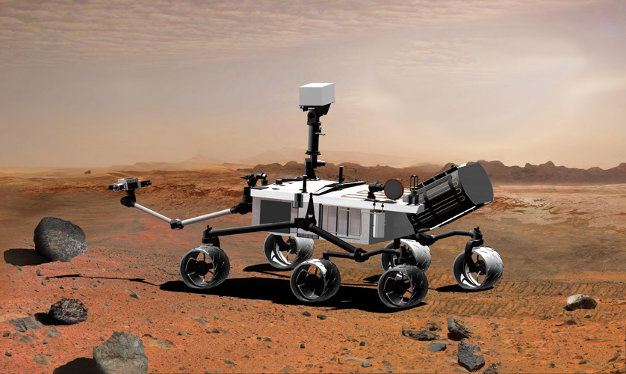
CS 46B

Spring 2023

Lab 3[[1]](#footnote-0)



In this lab you’ll start by improving the Mars Rover app from last week. Then you’ll do some experiments to help you get familiar with converting and casting. You might encounter some Exceptions along the way.

# Exit Interview

To receive credit for this lab, your group will complete an exit interview. To get an idea of the kinds of questions that will be asked look at the quiz you take at the beginning of the lab and the questions highlighted in blue that you encounter as you complete the lab instructions.

# Updating the DamagedRover class

In this part, you will improve on the Mars Rover code from last week. Use your own solution if it worked; or use the one included in this assignment. Write a short paragraph describing what the public void simulateStormDamageTravel() method in the DamagedRover class does. If you have questions about what the code is doing ask your lab instructor

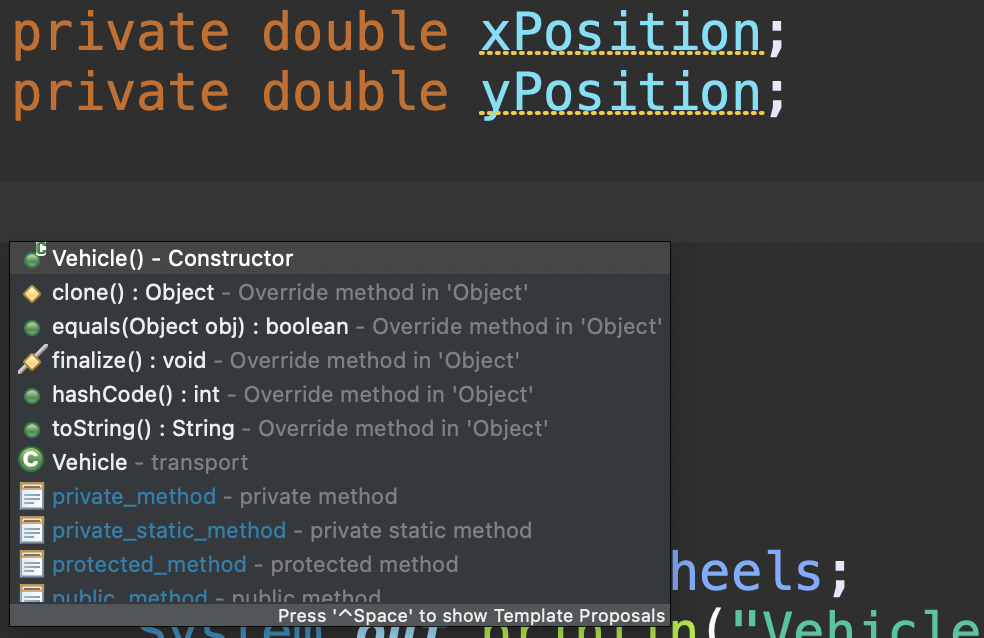
In last week’s lab, the DamagedRover class had a variable “position” to keep track of where it is with respect to where it got dropped by the Martian storm. That’s poor design. *Any* vehicle should know where it is. Refactor (change) your code so that it does what it did last week, but with a better design. Think about why these changes improve your code.

## Updating a superclass (Vehicle.java)

Add two instance variables to the Vehicle class

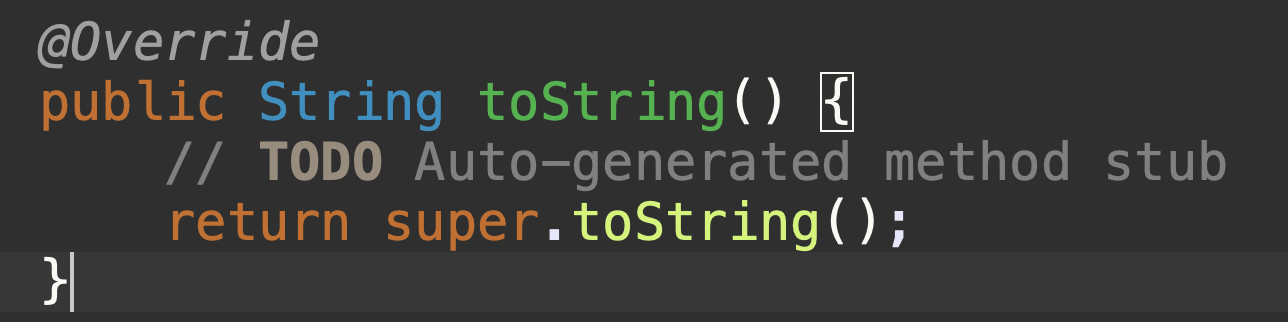
* private double xPosition
* private double yPosition

Now let’s see a neat feature of Eclipse that will save you time programming. Move the cursor below the closing brace, }, of the Vehicle constructor and hit Ctrl-Space. You’ll get a window similar to the image below.

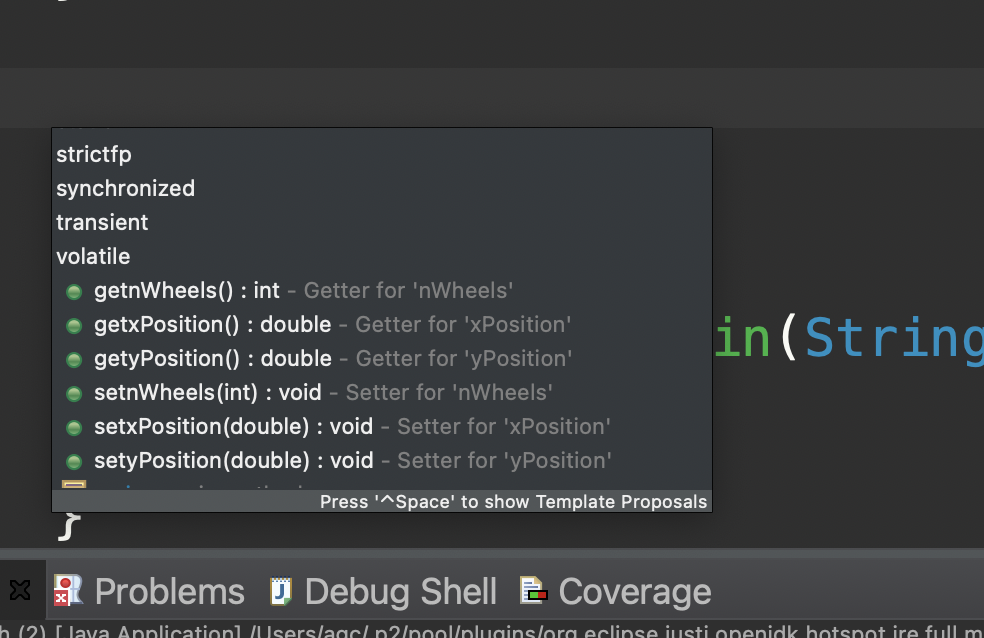


This will let you easily add a constructor and basic methods. Unfortunately, it will only add the default (no arguments) constructor, and we needed a constructor that takes in an argument to set the nWheels instance variable. You completed this during the last lab.

As you know from CS46A, it is important to have toString methods in the classes you create so you can print out meaningful information about instances of that class. Use CTRL-Space to bring up the window again and click on the toString method. What happens? You should get something similar to the image below. What String is the initial method returning? Update the toString to return something meaningful.



Let’s add getters for our instance variables. Move your cursor below the closing brace of the toString method and hit CTRL-Space. Scroll down (see image below) and select getXPosition. What happens? Repeat the process for getYPosition.



Remember how tedious it was in Lab 1 to add the getters and setters to the Planet class? Now you know a shortcut to add them to all your classes quickly and with a much lower probability of making a mistake.

Another Eclipse trick is whenever you type anything, save your work. This forces the compiler to run. To save, type CTRL-S in Windows, CMD-S in MacOS.

Now add two more methods to the Vehicle class

* public void setPosition(double xPosition, double yPosition), which sets those instance variables.
* public void changePositionBy(double xDelta, double yDelta), which adds xDelta to xPosition and adds yDelta to yPosition. The deltas might be negative.

## Updating the DamagedRover Class

We are going to update the DamagedRover class to take advantage of the refactor of the Vehicle class. Remember, Vehicle is a superclass of DamagedRover so instances of DamagedRover have access to all of Vehicles methods.

Make the following changes

1. Delete the position instance variable. Why don’t we need this anymore?
2. At the start of simulateStormDamageTravel, call setPosition(0, 0). We get to define distance with respect to any origin we like, so let’s decide that (0, 0) is the place where the storm dropped the rover.
3. Add method private void move(double distance, boolean forward), which moves the rover a positive (if forward is true) or negative (if forward is false) distance in the X direction.
   1. Remember the rover’s steering is damaged, so it can't move in the Y direction.
   2. This method should call changePositionBy, which DamagedRover inherits.
   3. How can you use changePositionBy to only move in the X direction?
4. Modify your simulateStormDamageTravel method to use the new methods.

Run DamagedRover as an app, and verify that you get the same results as last week. Is Rover likely to fall off a cliff before it runs out of power? Are you surprised by this result? Why?

Why is this design better than the original design?

# Debugging DamagedRover

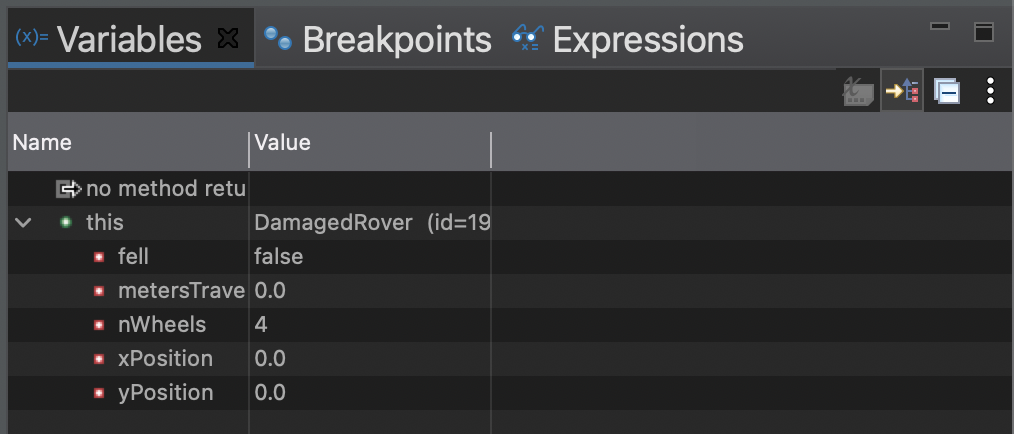
We are going to use the Eclipse Debugger to look at the code in the while loop in the simulateStormDamageTravel method. Debugging Mode. If Eclipse isn’t displaying source code line numbers, right-click in the margin to the left of the code and select “Show Line Numbers”.

Create a breakpoint in the first line of the while loop in the simulateStormDamageTravel method. Right click on the line number for that line, and select “Toggle breakpoint”. ***Describe what you see. Make sure everyone in your group is seeing something similar.***

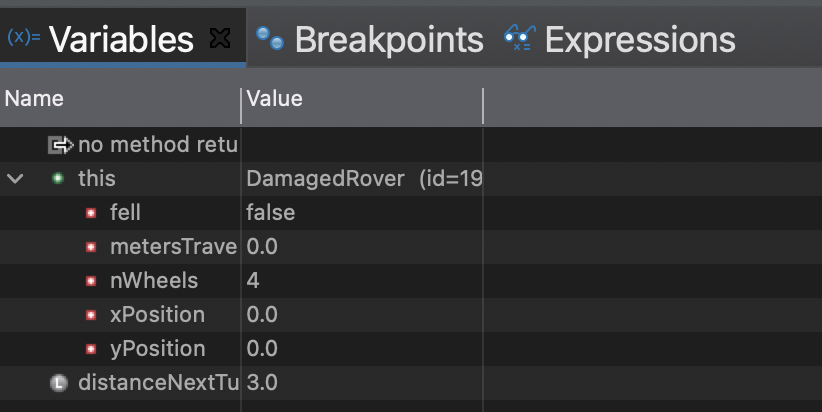
Execute up to the breakpoint. Instead of clicking the Run button, click the Debug button, which looks like a green bug and is near the Run button. You will probably get a “Confirm Perspective Switch” dialog. Select “Remember my decision” and then click “Yes”. Your Eclipse will rearrange itself. The JVM has executed your code up to the breakpoint that you set. The current line is marked in the main source code window.

In your Eclipse window, you should see something that looks like this. It’s on the right hand side of my debug perspective. The image below has been rotated 90 degrees. Click the icon that looks like (x)= 

That should bring up a window that looks like this. Notice that I clicked on the down arrow to expand this -> the instance of DamagedRover that is calling the simulateStormDamageTravel method. What do you notice?



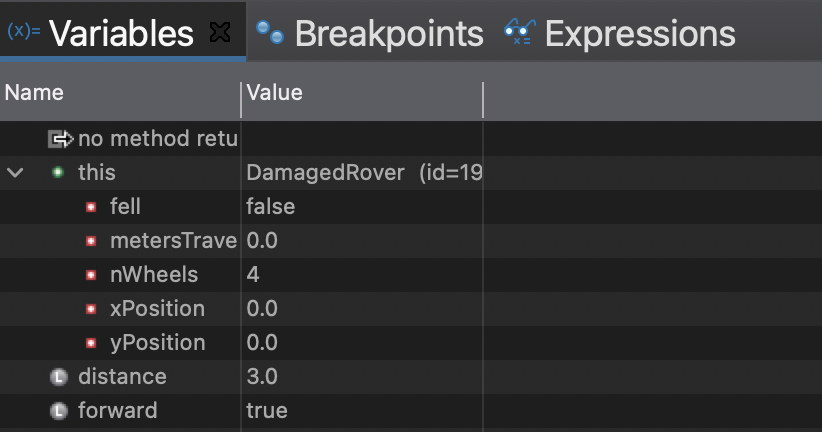
Now click which is going to Step Over the current line (this will execute the code on that line without stepping into the Math.random() function). Your Variables Window now might look something like this. What changed? Make sure everyone in your group is at the same point.



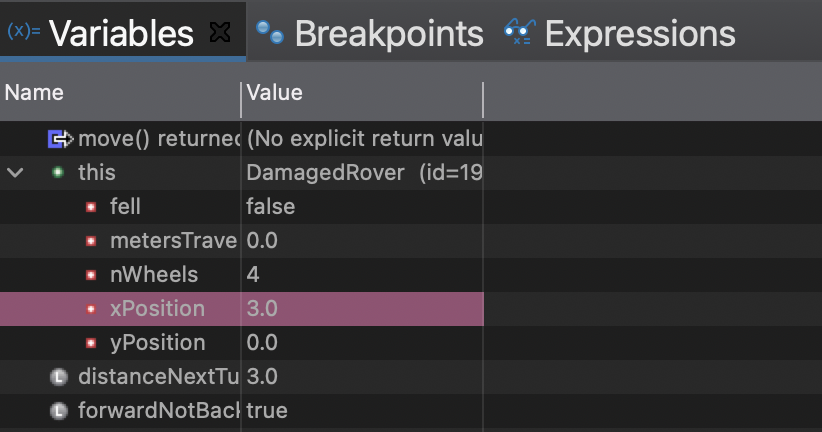
Now hover your mouse cursor over any occurrence of the variable distanceNextTurn in that window. What happens?

Now click the Step Into button: . You are now most likely in the Math class. You don’t want to debug the Math class. Press the Step Return button, , to return to your code. Now Click the Step Over button, , to move onto your call for the move function. This is a function you just wrote, let’s Step Into, , it and see what happens.

What happened when you stepped into the move function? Notice how your Variables window changed. It probably looks something like this now. What changed? Why do you think it changed?



What happens if I press the Step Into button, , again? Where do I end up? Once you feel like you have a good understanding of what step into does, press the Step Return button, , to get you back to the simulateDamagedRover method (this may be more than one click). Notice that the value of xPosition has changed.



Now press the button, . What happens?

To run your program to completion press, , to skip all the breakpoints. And press  again. Change from the Debug Perspective to the Java Perspective by clicking this button, , which is probably in the upper-right corner.

## Recap

We learned how to look at the values of local variables and the values of instance variables. We also learned how to create a breakpoint that let’s us stop the code at a certain point and run through the program more slowly looking at the values of the variables. This can help us avoid putting a bunch of println statements in our code that we then have to remove. We can navigate through the code in three possible ways.

Step Into: Step into the function executed on the current line. If the current line doesn’t represent a method call, this button simply skips the next line.

Step Over: skip the code of the function on the current line you are debugging and move immediately to the next line.

Step Return: skip the rest of the code in the function

You now have basic Eclipse survival skills. During the semester, it is your responsibility to get familiar with more features of Eclipse. Sometimes I will show a feature in class or your lab instructor might show you. There are also lots of resources on the internet about how to use Eclipse. Being familiar with Eclipse will make the programming components of this class simpler.

# Converting and Casting with primitives

In the second half of this lab you will experiment with converting and casting among 6 numeric primitive types: byte, short, int, long, float, and double. Recall that the rules about when casting is required are based on which of the 2 types involved is “wider” than the other. Here the “wider” type is not the type with more bits; it’s the type with the broader range.

In your current Eclipse project, under src, create a package called cast. In that package, create a class called Primitives.

Add a static method to Primitives called dumpMaxValues(), which prints out the maximum value for each of the 6 numeric primitive types. To determine the maximum value for any type, look up the API page for its wrapper type (class [java.lang.Byte](https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Byte.html), [java.lang.Short](https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Short.html), etc.), and go to the “Fields” section; you’ll find a static defined constant that equals the max value. Wrapper classes let you use primitive data types as Objects.

In dumpMaxValues(), print that constant for each type. Write a main() that just calls dumpMaxValues(). *Note: the minimum value for each type is approximately the negative of the maximum*. Based on the output of dumpMaxValues(), List the 6 types in order from narrowest to widest. Remember that “narrow” and “wide” don’t describe a type’s number of bits, they describe its numerical range.

***Before continuing, here’s some terminology. In an assignment statement like x = y; we say that x is the “left-hand side” or “LHS” of the statement, and y is the “right-hand side” or “RHS”.*** Now do some experiments, trying various LHS and RHS primitives. Create a table like the one below.

| LHS Type | RHS Type | RHS Value | Wider type (LHS or RHS) | Assignment needs cast? (yes/no) | Does assignment truncate or round? | Is LHS **exactly** RHS, **approximately** RHS, or **different** from RHS? |
| --- | --- | --- | --- | --- | --- | --- |
| int | long | Max long |  |  | n/a |  |
| int | long | Max long - 5 |  |  | n/a |  |
| long | int | Max int |  |  | n/a |  |
| double | byte | 100 |  |  | n/a |  |
| byte | double | 45.67 |  |  |  |  |
| byte | double | 456.789 |  |  |  |  |
| long | float | 12345.6789f |  |  |  |  |
| long | float | Max float |  |  |  |  |
| float | long | Max long |  |  | n/a |  |

For each row in the table, write 4 lines in main() like the example below. These lines should be:

1. A comment.
2. An assignment to a variable of the RHS type.
3. An assignment of the variable in the line above, to a variable of the LHS type. You might need a cast here.
4. A println that shows the values of the 2 variables.

For example, your code for the 1st row might be the following:

// int = long;

long l = Long.*MAX\_VALUE*;

int i = (int)l;

*System.out.println*("long to int: " + l + ", " + i);

The “truncate or round” column is for rows where you assign a type that has value to the right of the decimal point, to a type that doesn’t. “Truncate” means to ignore the value to the right of the decimal point. Determine which operation Java uses. For the last column, enter one word for each row: “exactly”, “approximately”, or “different”.

# Converting and Casting with Objects

Now you will do some experiments with converting and casting with objects and references.

1. In src, create a new package called “birds”.
2. Create a public interface called Flyer with one method, public void fly();
3. Create a public class Bird that implements the Flyer Interface.
   1. If you need a reminder about interfaces, look back to the lecture 4 notes.
   2. The fly() method should print “flap flap” to the console.
4. Create a public class Duck that extends Bird.
   1. Add a public instance method called quack() that prints out “Quack quack” to the console.
5. Create a public class Swan that extends Bird.
   1. Add a public instance method called glide() that prints out “I’m Graceful” to the console.
6. Add the following main method in the Swan class.

public static void main(String[] args) {

Bird duck = new Duck();

Swan s = new Swan();

Bird b = s;

Duck d = b;

Duck d1 = duck;

d1.quack();

d.quack();

}

* 1. If you get any compiler errors in assignment lines, fix them by casting. For every line that you fixed by casting, write “Fixed by casting: RHS type = xxxx, LHS type = yyyy, RHS is zzzzzz of LHS”. For zzzzzz, write “subclass” or “superclass”. Check this answer with your lab instructor.
  2. Do you need to cast when the RHS is a superclass of the LHS? Do you need to cast when the RHS is a subclass of the LHS?
  3. Look at Swan’s main() method. Discuss what you think will happen when the line “d1.quack();” executes. What is the class of the object being asked to quack? Does it have a quack() method?
  4. Look at Swan’s main() method. Discuss what you think will happen when the line “d.quack();” executes. What is the class of the object being asked to quack? Does it have a quack() method?
  5. Now run Swan as an application.
     1. What happened when the line “d1.quack();” executed?
     2. What happened when the line “d.quack();” executed?

*Here is another example of encountering Exceptions. Here’s the main point to take away for now: When you cast with primitive types, the risk is that the RHS value might not fit into the new type, and the value might get badly distorted. When you cast with objects and references, the risk is that the new type might not be compatible with the class of the actual object, causing the JVM to throw a ClassCastException.*

1. Modified from material provided by Dr. Philip Heller and Dr. Cay Horstmann and Dr. Chakarov [↑](#footnote-ref-0)